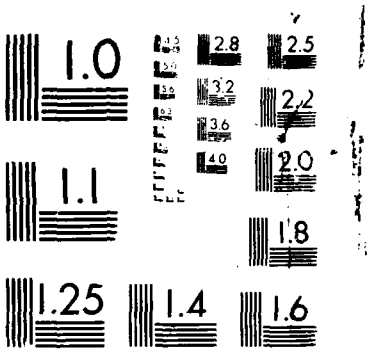
AD-A195 167 SPACE-EFFICIENT QUEUE MANAGEMENT USING FIXED-CONNECTION NETWORKS..(U) MASSACHUSETTS INST OF TECH CAMBRIDGE MICROSYSTEMS RESEARCH CE.. T LEIGHTON ET AL. NOV 87 1/1

UNCLASSIFIED VLSI-MEMO-87-426 N00014-80-C-0622 F/G 12/4 NL



VLSI Memo No. 87-426
November 1987

SPACE-EFFICIENT QUEUE MANAGEMENT USING FIXED-CONNECTION NETWORKS

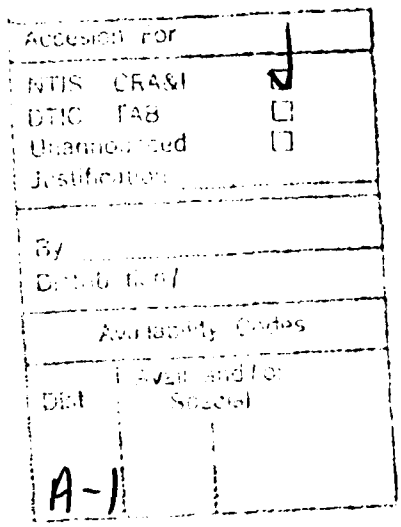
Tom Leighton and Eric Schwabe

Abstract

One of the main difficulties in designing algorithms for large scale parallel machines is making sure that the capacities of the local memories are not exceeded. In this paper, we present a general scheme for dynamically reorganizing memory so that local memory constraints are never exceeded provided that global memory constraints are not exceeded. The scheme is simple, real-time, space-efficient, deterministic and transparent to the programmer. It requires only that the total hardware used (i.e., wires and total memory) exceed the number of local memories by a logarithmic factor. In return, the scheme guarantees an arbitrarily high percentage utilization of the total memory, independent of whatever local demands for memory arise. We analyze the behaviour of our scheme in worst-case and average-case settings, and we show that it is optimal in many respects, even when compared to randomized algorithms.

Microsystems Research Center Room 39-321 | Massachusetts Institute of Technology | Cambridge Massachusetts 02139

Telephone (617) 253-8138



Acknowledgements

This work was supported in part by the Air Force under contract number AFOSR-86-0076, the Defense Advanced Research Projects Agency under contract number N00014-80-C-0622, the MIT Army Center for Intelligent Control, and an NSF Presidential Young Investigator Award with matching funds from AT&T and IBM.

Author Information

Leighton and Schwabe: Department of Mathematics and Laboratory for Computer Science, MIT, Cambridge, MA 02139; Leighton: Room 2-377, (617)253-3662; Schwabe: Room NE43-328, (617)253-1365.

Space-Efficient Queue Management Using Fixed-Connection Networks

Tom Leighton*
Eric Schwabe†
MIT Department of Mathematics
and Laboratory for Computer Science
Cambridge, MA 02139

Abstract

One of the main difficulties in designing algorithms for large scale parallel machines is making sure that the capacities of the local memories are not exceeded. In this paper, we present a general scheme for dynamically reorganizing memory so that local memory constraints are never exceeded provided that global memory constraints are not exceeded. The scheme is simple, real-time, space-efficient, deterministic and transparent to the programmer. It requires only that the total hardware used (i.e., wires and total memory) exceed the number of local memories by a logarithmic factor. In return, the scheme guarantees an arbitrarily high percentage utilization of the total memory, independent of whatever local demands for memory arise. We analyze the behaviour of our scheme in worst-case and average-case settings, and we show that it is optimal in many respects, even when compared to randomized algorithms.

1 Introduction

In very large scale parallel computers, memory is distributed in small chunks among a large number of processors. Typically, the total memory of the system is quite large, but each local memory is quite small. Insuring that the capacity of the local memories is not exceeded is one of the main problems in designing parallel algorithms and architectures. Indeed, it is often a relatively simple matter to insure that total memory capacity is not exceeded, but fluctuations in the demand for local resources invariably arise that make allocation of memory in certain "hot spots" much more difficult to handle.

As an example, consider the problem of routing a permutation of N packets on an N-processor machine. At the beginning and the end of the routing, each processor has precisely

*supported by Air Force Contract OSR-86-0076, DARPA Contract N00014-80-C-0622, the MIT Army Center for Intelligent Control, and an NSF Presidential Young Investigator Award with matching funds from ATT and IBM

†supported by a National Science Foundation Graduate Fellowship

one packet. Depending on the algorithm used to route the packets, however, large numbers of packets might bunch up at a few nodes during intermediate stages of the routing. For example, up to $\Omega(\sqrt{N})$ packets might bunch up at a single node if an oblivious routing strategy is used [BH]. Although we could handle the local space crunch by allocating $\Theta(\sqrt{N})$ capacity to each local memory, it would be terribly wasteful to do so since we would only be using a total of N out of $\Theta(N^{3/2})$ space overall at any point in time.

A variety of approaches are used in practice to try to overcome the problems associated with localization of memory. These approaches include: allocation of "extra" space at each processor, misallocation (i.e., sending data where it doesn't want to go, but where there is space), destruction of overflow data, freezing movement of data until space becomes available, and randomizing the desired location of data by (for example) hashing the memory on a global basis. Each of these techniques has its advantages and disadvantages. Probably the most popular is randomization which has been applied quite successfully to packet routing problems [VB] [R], but even here there is no iron-clad guarantee of success, and operations such as hashing the entire memory involve a great deal of overhead. Moreover, randomization has only been shown to work for certain very specific problems.

In this paper, we present a general scheme for automatically reorganizing memory so that local memory bounds are never exceeded provided that global memory bounds are not exceeded. The scheme is real-time, space-efficient, deterministic, transparent to the programmer, and perhaps best of all, quite simple. It requires only that the total hardware used (i.e., wires and total memory) exceed the number of local memories being simulated by a logarithmic factor. In return, the scheme guarantees an arbitrarily high percentage utilization of the total memory, independent of whatever local demands for memory arise.

Our model assumes that each local memory can be treated as a linear array that is accessed through a *port* associated with a single processor in the network. This is somewhat restrictive in that we do not allow a processor to have random access to its local memory, but it is general enough to include memories that are queues, stacks, priority queues, and the like. We break up memory and data into fundamental units called *packets*. We assume that one packet can traverse a single wire in a single unit of time. Packets can be thought of as bits, bytes, or entire blocks of data. The arrival and departure of data packets through a port is governed by a parameter λ that is usually assumed to be a constant. In particular, we assume that no more than $\lceil \lambda t \rceil$ packets can arrive or depart through a port in t consecutive steps. Lastly, we let m denote the number of local memories, and p denote the total number of packets in all the memories. Note that the p packets may or may not be distributed evenly among the m processors. Indeed, all p packets may be contained in a single local memory.

The task facing us is to design a network and algorithm to simulate any action of m local memories subject to the preceding constraints. To effect the simulation, we will construct an N-node degree-d network where each processor has a local memory of size q. Of course, if q is as large as p, the simulation is trivial. The object is to make N, d and q as small as possible relative to m, λ and p. In this paper, we describe a butterfly-based construction for which $\lambda = \Theta(1)$, $d = \Theta(1)$, $q = \Theta(1)$ and $N = \max(p, m \log m)$, and a hypercube-based construction for which $\lambda = \Theta(1)$, $d = \Theta(\log N)$, $q = \Theta(\log N)$ and $N = \max(m, p/\log m)$.

Both constructions are optimal in the sense that the total storage needed for the simulation (Nq) is only slightly larger than the trivial lower bound of p when p is large, and in the sense that the simulations are real time. The butterfly-based construction is also optimal in the sense that any bounded-degree simulator must have $\Omega(m \log m)$ total memory, even if p is small and even if randomized algorithms are used. Whether or not a simulator can be constructed with fewer nodes and wires (say m) but with more local memory per node (say $\log m$) when $p = m \log m$ remains an open question. We show that such a simulator does exist when $p = m$, but this result is less interesting since it is not space-efficient.

Our simulations are based on an efficient solution to a specialized routing problem that we call *isotone routing*. In particular, an *isotone routing problem* is one for which the relative order of the elements being moved is unchanged. For example, the mapping $\{1 \rightarrow 3, 2 \rightarrow 4, 6 \rightarrow 5, 7 \rightarrow 8\}$ is an *isotone partial permutation*. We suspect isotone routing problems will turn out to be useful in other applications as well. As a simple example, we show how to use our isotone routing algorithm to route any permutation problem of size N on an $N \log N$-node butterfly in $O(\log^2 / \log \log N)$ steps, slightly improving the previous best known deterministic bound of $\Theta(\log^2 N)$

The remainder of the paper is divided into four sections. Our memory reallocation scheme and related constructions are presented in Section 2. Section 3 contains the lower bounds and proofs of optimality. Average-case results are dicussed in Section 4. Here it is interesting to note that although randomized algorithms fare no better than deterministic algorithms on worst-case problems, an average-case problem is substantially easier to handle than a worst-case problem. We conclude with some open questions and subjects of ongoing research in Section 5.

Due to the constraints on length, proofs and descriptions of constructions are severely abbreviated. We will also restrict our attention to the special case when each local memory is a simple queue.

2 Constructions

2.1 Isotone Routing

The building block of the space-efficient queueing networks discussed in this section is a butterfly-based network which can route a special class of partial permutations of $\{1, 2, \ldots, m\}$ deterministically and on-line in $\Theta(\log m)$ steps. The network in Figure 1 (for $m = 8$) can perform such routing for *isotone* partial permutations; we say that a partial permutation of $\{1, 2, \ldots, m\}$ is isotone if the mapping from sources to destinations is strictly isotone (since we are dealing with a partial permutation, this mapping is injective, so that the mapping being isotone is equivalent to its being strictly isotone). Note that the middle row of columns is not really necessary (it is included to simplify the algorithm description), and that this network, aside from the long vertical edges, is just a column permutation of the Benes back-to-back butterfly network.

The network can route an isotone partial permutation of $\{1, 2, \ldots, m\}$ as follows. The upper butterfly is used to count the sources, and to assign to each source its position in the left-to-right ordering of sources (0 for the leftmost source, 1 for the next...); this can be done using a prefix calculation, and an extra step to pass the values back to the top — a total of $\log m + 1$ steps. Then in $2 \log m$ steps, each source can be routed to the position in the middle row determined by this value and then on to its destination without collisions. That there are no collisions comes from the fact that, since we are packing all the sources to the left, we are in effect performing two instances (one in reverse) of deterministic on-line 0-1 routing (given 0's and 1's distributed among the sources, route them so that all the 0's are to the left of all the 1's), for which it is known that no collisions occur for routing using straightforward bit-flipping. Thus the isotone partial permutation is routed in $3 \log m + 1$ steps.

Note that if the sources are the same for a sequence of j isotone partial permutations (or even if each set of sources is included in the previous set), then we can use the results of the first prefix computation to route all j permutations one after the other, for a total time of $(\log m + 1) + (2 \log m + j - 1) = j + 3 \log m$. In particular, when each member of the set of sources $s_1 < s_2 < \ldots < s_i$ has some discrete interval of destinations of length at most j ($\{d_i, d_i + 1, \ldots, d_i + j_i - 1\}$, $j_i \leq j$), where for all k, $d_k + j_k - 1 < d_{k+1,1}$ (all the destinations of s_k are less than all the destinations of s_{k+1}), the above conditions are satisfied, and the routing can be accomplished in $j + 3 \log m$ steps.

Note also that by 'folding up' the two butterflies in the network, and identifying the resulting top and bottom rows of processors, we could reduce the size of the network to $m \log m$, and allow it to perform isotone routing in $3 \log m$ steps. However, this would eliminate any possibility of pipelining as described above, making the current configuration a better choice for the queueing algorithm, which relies heavily on such pipelining. Also, adding another set of long edges across the lower butterfly will allow us perform such routing in both directions on the network.

2.2 Queueing Networks and Algorithms

The queue management problem can be solved for $q = \Theta(1)$, $N = \Theta(m \log m)$, $\lambda < 1$ and p an arbitrarily large constant fraction of qN on a butterfly-based network.

In its most basic form, this network consists of m linear arrays of processors of size $C \log m$ (discussion of the C and the other constants involved in the construction, which affect the values of λ for which the algorithm will be correct, is omitted here), each with one of the m ports at one end, and at the other end a connection to a bidirectional isotone routing network. At the bottom of the isotone routing network are m linear arrays of size $\log m$. For example, see Figure 2.

The idea is that each of the m queues has some number (between $(u - v) \log m$ and $(u + v + 1) \log m$) of its elements stored in its upper linear array, and the excess stored in some range of the lower linear arrays. The algorithm cycles repeatedly, each time adjusting the number of elements in each upper linear array by either sending some multiple of $\log m$ queue

elements down to the lower linear arrays, or bringing up some multiple of $\log m$ elements from the lower linear arrays.

Each cycle consists of four phases (presented here in order of execution; in an enhanced version of the network and algorithm, further parallelism is used to sharpen results):

1. Overhead, in which it is determined for each processor whether queue elements will be sent down or retrieved, and its new range of lower linear arrays is calculated.

2. Retrievals, where all queues which have become 'too small' simultaneously bring elements from their ranges of lower linear arrays back to their upper linear arrays, using isotone routing.

3. Shifts, where those elements remaining in the lower linear arrays are moved to their new positions (as calculated in the Overhead phase), in preparation for new elements to be sent down.

4. Sends, where all queues which have become 'too big' simultaneously send elements form their upper linear arrays down to their ranges of lower linear arrays.

Analysis of the time required by these cycles of the algorithm gives us constraints on the values of λ for which the algorithm will work in terms of u, v, and C, and allows us to find the optimal values of u and C for a given v. For the basic network and algorithm, we will always have $\lambda < \frac{1}{2}$, but we also describe an enhanced version with a higher degree of parallelism which allows us to get λ arbitrarily close to 1 by choosing sufficiently large v. Although to simplify the explanation, this algorithm is explained in terms of managing stacks, adjustments can be made to allow us to simulate general linear arrays (i.e., stacks, priority queues, FIFO, etc.).

We also describe a hypercube-based algorithm, obtained by identifying entire columns of the butterfly-based network into single nodes. This network solves the problem with $q = \Theta(\log m)$ and $N = m$.

In their paper on the token distribution problem, Peleg and Upfal describe an n-node bounded-degree network which can, with $\Theta(\log n)$ local storage, solve routing problems for up to n packets in $\Theta(\log n)$ time, as long as no source or destination node has multliplicity greater than $\log n$. One might try replacing the isotone routing network with an m-node network of this type and transforming each sub-array of $\log m$ nodes in the linear arrays into a single node with $\log m$ size internal storage, yielding a bounded-degree network with $q = \Theta(\log m)$ and $N = \Theta(m)$ to solve the queueing problem. However, since the routing network has no pipelining abilities, we would have to restrict the total number of packets in the network to be $N = m$, only using $\frac{1}{\log m}$ of the available space.

2.3 An Application of Isotone Routing to General Permutation Routing

There is a deterministic algorithm for routing a partial permutation on an n-node hypercube in $\Theta(\frac{\log^2 n}{\log \log n})$ steps [K] [L] which, using isotone routing, yields a partial permutation routing algorithm on the butterfly which runs in the same number of steps.

The hypercube algorithm goes as follows: Route greedily for the first $\log\log n$ bits. At this point, each $\log n$-node subcube defined by a setting of these $\log\log n$ bits can contain at most $\log n$ packets. Using isotone routing we can spread out these packets in their respective subcubes, at most one to each node, in $\Theta(\log n)$ steps; repeat this process for each subproblem. Thus the total running time is $\frac{\log n}{\log\log n}\Theta(\log n) = \Theta(\frac{\log^2 n}{\log\log n})$.

On the butterfly, the corresponding algorithm is as follows: route greedily for the first $\log\log n$ levels of the butterfly; then, as before, each sub-butterfly defined by a setting of these $\log\log n$ bits can contain at most $\log n$ packets. Thus we can use isotone routing to distribute these packets evenly within the sub-butterfly in $\Theta(\log n)$ steps; repeat this process on each sub-butterfly. As before, the total running time is $\Theta(\frac{\log^2 n}{\log\log n})$. The key difference is that we must simulate the $\log n$-size queues of the hypercube with constant-size queues contained in the corresponding nodes of the butterfly. The details are not difficult to work out.

3 Lower Bounds

Here we prove lower bounds on the space needed to solve the memory management problem, for both deterministic and randomized algorithms. In looking for lower bounds, we consider an *off-line* formulation of the problem where we are given a final queue length for each port in the network and the problem is to find m slot-disjoint queues of the appropriate lengths from the m ports in the network; nodes of the network have random access to their slots. (A *slot* is a single location in a local memory of the simulating network.) Since the general queueing problem is at least as hard as the off-line problem, any lower bounds found for the off-line problem will apply to the on-line problem.

It is immediate from the problem definition that $m \leq N$ and $p \leq qN$; also, we must have $q \geq \lceil\lambda\rceil \geq \lambda$, since there must be room in a port to hold all the packets which could arrive (or be requested) in one time step. In addition, for r = the length of the longest queue and T = the number of time steps, we must have $r \leq \lceil\lambda T\rceil$.

3.1 The Deterministic Case

Lemma 1D: Let a queue-finding algorithm on an N-node network with m ports be given. If $mr > qN$ and $6rd^T \leq p$, then there is some assignment of queue lengths to the m ports for which the algorithm will not find disjoint queues of the appropriate lengths in the network.

Sketch of Proof: Assume that $mr > qN$ and $6rd^T \leq p$. If these constraints are satisfied, then we can force two ports to select queues which intersect in some slot as follows:

Since the total number of nodes at distance no greater than T from a given node is at most $3d^T$, each port's choice of queue can depend on the values in at most $3d^T$ ports. For each of the m ports, consider the queue of length r chosen when all $3d^T$ of the ports within distance T are assigned the value r. The total length of these queues is mr; since $mr > qN$, we can choose two ports for which these queues intersect at some slot. We assign values to ports as follows: assign to the two ports chosen above and to each port within distance T of either of them the value r; this is permissible since it involves at most $6d^T$ ports, and $6rd^T \leq p$ (there

are enough packets to go around). Assign any extra packets arbitrarily. This assignment of values to ports will cause the algorithm to fail by choosing intersecting queues, thus proving the Lemma.

Lemma 2D: If $4qN \log d < \lambda m \log \frac{mp}{12qN}$, then there is no correct queue-finding algorithm.

Sketch of Proof: Assume that $4qN \log d < \lambda m \log \frac{mp}{12qN}$, and let a queue-finding algorithm be given. Then it can be shown $r = \lfloor \frac{qN}{m} + 1 \rfloor$ and $T = \lfloor \frac{\log \frac{mp}{12qN}}{\log d} \rfloor$ satisfy $r \leq \lceil \lambda T \rceil$ and the conditions of Lemma 1D. It follows that the queue-finding algorithm must fail on some input. Therefore no correct queue-finding algorithm exists.

The following theorem shows a lower bound on the size of a family of $N = N(m)$-node networks with m ports, with a total queue length of $p = p(m)$.

Theorem: Let $N = N(m)$, and $p = p(m) \geq (qN(m))^{\epsilon}$ for some $\epsilon \in (0, 1]$ and sufficiently large m. If $qN \log d = o(m \log m)$ then no family of N-node networks with correct queue-finding algorithms for p packets exists.

Sketch of Proof: Assume that the above conditions hold, and let a family of N-node networks be given. Suppose that this family had correct queue-finding algorithms. Since $qN \log d = o(m \log m)$, we have that for all $c > 0$ and for sufficiently large m, $qN \log d < cm \log m$. Choose $c_1 > 0$ such that for sufficiently large m, $c_1 \leq \frac{\lambda}{4}(\epsilon + (\epsilon - 1)\frac{\log \log m}{\log m})$, and choose $c_2 > 0$ such that $c_2 \leq 12^{\frac{1}{\epsilon - 1}}$ (if $\epsilon = 1$, choose any $c_2 > 0$).

For these constants and sufficiently large m, $4qN \log d < \lambda m \log \frac{mp}{12qN}$, implying that (by Lemma 2D) no correct queue-finding algorithm can exist for sufficiently large m. This contradicts that this is a family of networks with correct queue-finding algorithms. We conclude that no such family exists.

3.2 The Randomized Case

For the randomized lower bound we use the most general version of an off-line randomized algorithm, where there are an arbitrary number of public random bits which all processors can access and, as before, each port has a value which is its final queue length.

Claim: Suppose $mr > 3qN$. Then for any set of r-length queues from the m ports, there exist distinct ports $p_1, p_2, \ldots, p_{\frac{2m}{3}}$ such that for all $i \in 1, 2, \ldots, \frac{m}{3}$, the queues from p_{2i-1} and p_{2i} intersect in some slot.

Sketch of Proof: Since $mr > 3qN$, we have that $[m - 2(i-1)]r > qN$ for $i \in 1, 2, \ldots, \frac{m}{3}$. Thus for $i = 1, 2, \ldots, \frac{m}{3}$ we can find an intersecting pair of queues, and remove them and their ports from consideration. This will yield $\frac{m}{3}$ distinct pairs of ports whose queues intersect, as desired.

Claim: Suppose $mr > 3qN$. Then given any randomized queue-finding algorithm (and sufficiently large m), there is some set of $2m^{\frac{1}{2}+\delta}$ ports (for any $\delta \in (0, \frac{1}{2})$) for which, when the values of all ports looked at during the computation are r, we have that Pr(the algorithm fails for bit setting B) is exponentially close to 1.

Sketch of Proof: Suppose $mr > 3qN$, and let a randomized queue-finding algorithm and a setting of the random bits B be given.

We consider the collision graph G, whose nodes are the m ports, and for which there is an edge between x and y in G if and only if the queues from ports x and y intersect when all values seen in other ports are r, and the random bits are set according to B. By the previous Claim, there is a set of $\frac{m}{3}$ independent edges (edges with no common endpoints) in G.

We then show that for a random set of $2m^{\frac{1}{2}+\delta}$ ports, the probability that the induced subgraph of the collision graph contains at least one of these edges is exponentially close to 1. This means that for almost all sets of $2m^{\frac{1}{2}+\delta}$ ports, if all the ports looked at by any of these ports had value r, then the algorithm would choose some pair of intersecting queues for bit setting B. Therefore for each possible bit setting B, all but exponentially few of the possible sets of $2m^{\frac{1}{2}+\delta}$ ports cover an edge in the collision graph, so that there must be some set of $2m^{\frac{1}{2}+\delta}$ ports which covers an edge in the collision graph for all but exponentially few bit settings. This is the set required by the Claim. In fact, the same property holds for all but exponentially few of the sets of $2m^{frac12+\delta}$ ports.

From this point, the argument follows the same program as the proof of the deterministic lower bound.

Lemma 1R: Let a randomized queue-finding algorithm on an N-node network with m ports be given. If $mr > 3qN$ and $6m^{\frac{1}{2}+\delta}rd^T \leq p$, then there is some assignment of queue lengths to the m ports for which the algorithm will fail (that is, not find disjoint queues of the appropriate lengths in the network) with probability exponentially close to 1.

Sketch of Proof: Assume that $mr > 3qN$ and $6m^{\frac{1}{2}+\delta}rd^T \leq p$. Consider a set of $2m^{\frac{1}{2}+\delta}$ for which all but exponentially few of the possible settings of the random bits cause the algorithm to select intersecting queues when the values in all the ports looked at during the computation are r. As in the determinisitc argument, a port can learn the values in at most $3d^T$ ports in T steps.

Thus by assigning the value r to the $6m^{\frac{1}{2}+\delta}d^T$ ports which can be looked at during the course of the computation (and assigning any extra packets arbitrarily), we insure that for all but exponentially few of the possible settings of the random bits intersecting queues will be chosen. Thus for this assignment of values to ports, the algorithm will fail with probability exponentially close to 1.

Lemma 2R: If $12qN \log d < \lambda m \log \frac{m^{\frac{1}{2}-\delta}p}{36qN}$, then there is no randomized queue-finding algorithm for which we cannot find an input on which it fails with probability exponentially close to 1.

Sketch of Proof: Assume that $12qN \log d < \lambda m \log \frac{m^{\frac{1}{2}-\delta}p}{36qN}$, and let a randomized queue-finding algorithm be given. Then $r = \lfloor \frac{3qN}{m} + 1 \rfloor$ and $T = \lfloor \frac{\log \frac{m^{\frac{1}{2}-\delta}p}{36qN}}{\log d} \rfloor$ satisfy $r \leq \lceil \lambda T \rceil$ and the conditions of Lemma 1R. It follows that the randomized queue-finding algorithm must on some input almost certainly fail. Therefore no randomized queue-finding algorithm for which such an input cannot be found can exist.

The following theorem shows a lower bound on the size of a family of $N = N(m)$-node networks with m ports, with a total queue length of $p = p(m)$, which is within a constant factor of the lower bound already found for deterministic algorithms.

Theorem: Let $N = N(m)$, and $p = p(m) \geq (qN(m))^{\epsilon+\frac{1}{2}}$ for some $\epsilon \in (0, \frac{1}{2}]$ and sufficiently large m. If $qN \log d = o(m \log m)$ then no family of N-node networks with randomized queue-finding algorithms for p packets where we cannot find some bad input exists.

Sketch of Proof: Assume that the above conditions hold, and let a family of N-node networks be given. Suppose that this family had randomized queue-finding algorithms without bad inputs. Choose $\delta \in (0, \epsilon)$. Since $qN \log d = o(m \log m)$, we have that for all $c > 0$ and for sufficiently large m, $qN < cm \log m$. Choose $c_1 > 0$ such that for sufficiently large m, $c_1 \leq \frac{\lambda}{12}(\epsilon - \delta + (\epsilon - \frac{1}{2})\frac{\log\log m}{\log m})$; this quantity is positive for sufficiently large m, since $\epsilon > \delta$). Choose $c_2 > 0$ such that $c_2 \leq 12^{\frac{1}{\epsilon-\frac{1}{2}}}$ (if $\epsilon = \frac{1}{2}$, choose any $c_2 > 0$).

For these constants and sufficiently large m, $12qN \log d < \lambda m \log \frac{m^{\frac{1}{2}-\delta}p}{36qN}$, implying that (by Lemma 2R) no randomized queue-finding algorithm without a bad input can exist for sufficiently large m. This contradicts that this is a family of networks with randomized queue-finding algorithms free of bad inputs. We conclude that no such family exists.

It follows that, in either the determinstic or the randomized case, when the conditions of the theorem hold, we must have $qN \log d = \Omega(m \log m)$ in order for a family of bounded-degree networks with correct algorithms to exist. If local storage is constant, we must have $N \log d = \Omega(m \log m)$, so that the butterfly-based construction is optimal up to a constant factor. For $q = \Theta(\log m)$, the hypercube variation is not necessarily optimal due to is logarithmic degree.

4 Randomized Inputs

In order to study the effects of random inputs, we use a Poisson model of queue arrivals, where at each time step, the number of packets arriving (or being requested) at a port has a Poisson distribution with mean λ; note that this is a somewhat different and more general notion of arrival rate than was used before. Thus if K is the number of packets arriving or departing at a port during one time step,

$$Pr(K = x) = \frac{e^{-\lambda}\lambda^x}{x!}$$

for $x = 0, 1, 2 \ldots$.

We will show that a group of $\log m$ ports will only with very small probability accumulate more than a total of $\log m$ packets In this case, we can solve each of the $\frac{m}{\log m}$ instances of this smaller problem with $q = \Theta(\log\log m)$, for a total of $\frac{m}{\log m}\Theta(\log m \log\log m) = \Theta(m \log\log m)$ space. This is much less than is required in the worst case, even if randomized algorithms are allowed.

5 Ongoing Research

The butterfly-based construction in section 2 shows that the lower bounds in section 3 are tight (to within a constant factor) when local storage is constant. However, $N = m \log m$ for this construction. Can a $q = \log m$, $N = m$ network with bounded degree and efficient space usage be found, or can the lower bounds be tightened to rule out such a possibilty?

Extension of our constructions to simulate random access local memories is not possible without a degradation in simulation time (i.e., real-time simulations are no longer possible), but it might be interesting if the results could be extended to simulate local memories that act like trees, if this is possible. Another area of research is to find the best achievable behavior for other specific networks — for instance, how many ports and packets can be managed in an N-node mesh?

References

[BH] A. Borodin and J. Hopcroft, incomplete reference.

[K] B. Kuszmaul, personal communication, 1984.

[L] T. Leighton, Class notes in Theory of Parallel and VLSI Computation, 1984.

[PU] D. Peleg and E. Upfal, "The Token Distribution Problem," Proc. of 27th Ann. Symp on Foundations of Computer Science, 1986, pp 418-427.

[R] A. Ranade, "How to Emulate Shared Memory," Proc. of 28th Ann. Symp. on Foundations of Computer Science, 1987, pp 185-194.

[VB] L. G. Valiant and G. J. Brebner, "Universal Schemes for Parallel Computation," Proc. of 13th Ann. ACM Symp. on Theory of Computing, 1981, pp 263-277.

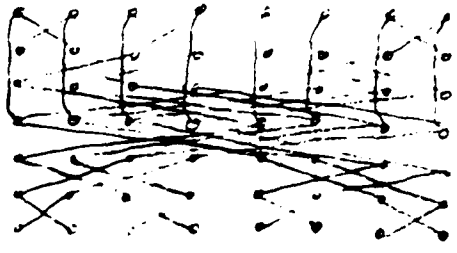


Figure 1

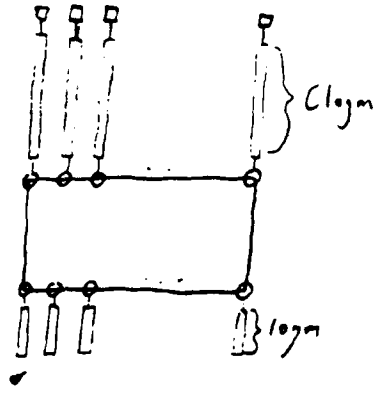


Figure 2

END

DATE

FILMED

8-88

DTIC